Rapidly Re-Configurable Flight Simulator Tools for Crew Vehicle Integration Research and Design

Summary of Research
Grant NAG 1-2175

submitted to
NASA Langley Research Center
Hampton VA 23681-2199

for the period
March 30, 1999 – December 31, 2000.

Attention: Paul C. Schutte and Anna Trujillo

Amy R. Pritchett, Sci.D., Principal Investigator
Schools of Industrial and Systems Engineering and Aerospace Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0205
(Tel) 404-894-0199
(Fax) 404-894-2301
Amy.Pritchett@isye.gatech.edu

Originally submitted December 2000
Re-submitted February 2002

ABSTRACT

While simulation is a valuable research and design tool, the time and difficulty required to create new simulations (or re-use existing simulations) often limits their application. This report describes the design of the software architecture for the Reconfigurable Flight Simulator (RFS), which provides a robust simulation framework that allows the simulator to fulfill multiple research and development goals. The core of the architecture provides the interface standards for simulation components, registers and initializes components, and handles the communication between simulation components. The simulation components are each a pre-compiled library 'plug-in' module. This modularity allows independent development and sharing of individual simulation components. Additional interfaces can be provided through the use of Object Data/Method Extensions (OD/ME). RFS provides a programmable run-time environment for real-time access and manipulation, and has networking capabilities using the High Level Architecture (HLA).

INTRODUCTION

Simulation is an integral part of aerospace research and design. Its ability to predict complex system behavior makes it valuable to the analysis and testing of many entities, including vehicles, on-board components such as avionics systems, pilot-interactive systems such as cockpit displays, flight control systems, and procedures for operation of vehicles[1].

Simulation can fit into all stages of research and design. During basic research and conceptual design, low- and medium-fidelity simulations can highlight fundamental issues and constraints on system design. As the design progresses, higher-fidelity models can be added to the system so that its output provides increasingly detailed and accurate feedback to designers. At the end of design, high-fidelity simulations can serve as a complement to (or replacement for) flight tests[2], and can serve to train the pilots who will first fly the aircraft.

However, despite the theoretical utility of simulations throughout design, the time and resources required to develop simulation software for research and design projects (henceforth referred to as design projects) can sometimes limit or prohibit their use. The development of simulation software can take a great deal of time and resources, to the extent that 'rapid' development has been described as that achievable in weeks to months[3]. Likewise, the development of simulations from scratch requires personnel with substantial skills in many areas, including software engineering and computer programming, computer graphics, and dynamic modeling.

Lacking the resources to develop a tailor-made simulation (and can not bide the delay it would cause), design projects commonly re-use already-existing simulation software[4]. Existing components may be modified, and existing simulations may have new components added to provide new functionality.

However, the impact of these modifications is often to make the software unusable for future projects or for subsequent stages of the design. Unlike 'traditional' simulation projects (which are focused on providing simulation software of high quality), design projects are often not motivated to maintain software standards, with result that the software quality can degrade. This concept is shown notionally in Figure 1.

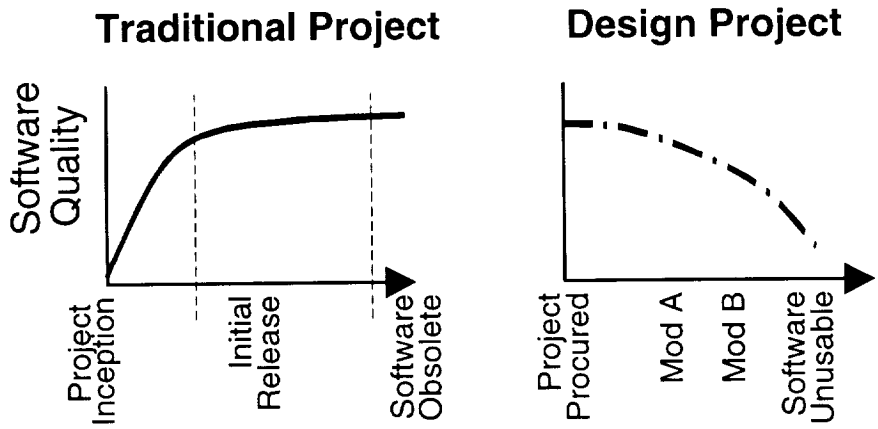

Figure 1. Time History of Simulation Software Quality in 'Traditional' Software Projects, v.s. Design Projects Where Simulation Is a Tool

Currently, there is no widely available architecture for simulations intended specifically for a comprehensive range of aerospace research and design needs. Solutions are often oriented towards a single objective and are difficult to apply or modify to fit additional requirements; for example, pilot-in-the-loop simulators can be difficult to convert to simulations capable of running fast-time analyses of flight control systems, and vice versa. Solutions that do provide a wide-ranging and robust framework are proprietary, expensive, platform-specific or difficult to obtain.[5]

The remainder of this paper discusses the design of simulation software meeting the needs of aerospace researchers and designers. First, these needs are reviewed. Then, object-oriented programming (OOP) and Object-Oriented Analysis and Design (OOAD) principles are introduced as mechanisms for developing a suitable framework. Based on this discussion, the development of the Reconfigurable Flight Simulator (RFS) is outlined.

REQUIREMENTS OF SIMULATION FOR RESEARCH AND DESIGN

Simulation may be applied in many different ways in aerospace research and design. Conceptual vehicle design may use lower fidelity simulations of single-aircraft flight dynamics as a tool to analyze vehicle sizing and configuration; human factors experiments may use mid-fidelity pilot-in-the-loop simulations; avionics prototyping and design may use simulation as a hardware-in-the-loop testing mechanism; design of operational procedures or mission scenarios may use batch simulations of multiple vehicles; flight control design may use fast-time simulations with high fidelity vehicle dynamic models; and flight tests require a flight simulator with fidelity high in every measure.

This range of needs highlights the need for a design project to have a simulation tool available that is *flexible*; i.e. the simulation should not be fundamentally constrained by its basic architecture to one mode of operation or to one level of fidelity.

A simulation for design projects must also accommodate user types that are fundamentally different than the users of current flight simulators. The first is that of the *general user*, who wants to use the simulation as part of his or her day-to-day design activities. The general user can be defined as a designer who does not want to interact with source code or recompile software. While all simulations must support general users, design projects are novel in that their general user is quite knowledgeable, and desires considerable power over the simulation so that he or she can reconfigure it and use it in a number of ways. With normal flight simulations, the general user is shielded from the underlying functionality through graphical user interfaces or simple configuration scripts[6]; in simulations for design projects, the general user benefits from greater power over the simulation.

Unlike most simulation software packages, simulations for design projects must provide support for an additional user class – that of the *developer*. In traditional software projects, developers are normally dedicated programmers and software engineers, whose primary purpose is to program the simulation. In design projects, the developer is often a member of the design team with some programming knowledge; the developer's purpose is to get the simulation running as a tool. In this case, the developer is motivated to get the simulation running quickly without any supervision of software quality. He or she may not be in a situation to understand the complete workings of all the simulation components, and therefore may not understand the impact of their modifications. Additionally, the developers in design projects may be distributed throughout an organization.

As such, simulation software for design projects must be designed to be inherently *robust* and *programmable:* robust in that modifications to one part of the software should not have widespread, unanticipated effects on other parts of the software; programmable in that the developer should find the overall framework of the simulation easy to understand, and he or she should quickly be able to find where and how modifications should be made to evoke the desired behavior.

To support design projects, simulations must be inherently *extendable*; i.e. it should be easy to add new functionality to the simulator through the addition of new components, rather than through fundamental changes to the entire architecture. Likewise, the components should be maintained in a form such that new components can be added to the simulation without destroying currently existing functionality.

Finally, a simulation will be the most suitable for wide-spread use if several practical considerations are met. For example, many design projects do not need simulations of such high-fidelity that they can not be run on Pentium-based personal computers, while other simulations may be so computationally expensive as to warrant investment in high-speed computers. As such, simulations that are restricted to one platform inherently limit their distribution. Likewise, simulations have more general utility when they do not require specific external hardware or peripherals.

OBJECT-ORIENTED ANALYSIS AND DESIGN

As programs have grown in complexity, Object-Oriented Programming (OOP) has been proposed to create software easy to design, modify and re-use[5]. OOP principles include: abstraction, inheritance, layering, encapsulation, and polymorphism.

Abstraction refers to the ability of OOP to define computational structures not as a sequence of logical functions, but as independent objects, each containing the functions (methods) and data required to perform the functions of that object. With this ability, the software engineer's design process relies on abstracting the overall behavior into individual objects, and then determining their functioning. Objects may themselves contain an inner hierarchical structure of lower-level objects.

Abstraction allows for the entire program to be viewed at the level of abstraction appropriate for the task. For example, a flight simulation architecture may, at a high-level of abstraction, be viewed as a collection of input-output objects, vehicle objects, and a timer object; conversely, a designer interested only in the lower-level task of improving the fidelity of a vehicle module can focus on that vehicle object alone, without needing to see the structure of the rest of the software.

Abstraction also provides a mechanism to structure the software using the same abstractions that are used by other domains. For example, an aircraft object could be broken down into sub-components in many different ways; a common choice is to mirror the different on-board systems (engines, flight controls, etc.). Various works in the literature have described suitable object definitions for flight simulators[7].

Inheritance refers to the ability, using OOP, to define base interface standards for the behavior of a type of object. For example, a base or parent 'aircraft' object might define what functions any aircraft object must be able to perform, and how its data can be accessed. Modules capable of simulating specific aircraft can then inherit from this base aircraft, providing specialization and additional functionality while guaranteeing that these modules will interact with the rest of the program like any object of type aircraft.

Once objects have been defined, OOP's capacity for *layering* can be applied; i.e. smaller, more primitive types of objects can be combined to create larger, more sophisticated objects. For example, an airport type might be made from a combination of runway, tower, hangar and taxiway objects.

Encapsulation is a mechanism by which low-level details about an object are 'hidden' within that object type, so that the entire program does not need to have access to, or work on, the low-level variables internal to an object. For example, an aircraft object type might be required to provide other objects with useful variables such as position, velocity, etc., while variables useful only to the aircraft dynamic model (such as stability derivatives) are encapsulated to stay within the aircraft type.

Polymorphism is an OOP mechanism by which objects inheriting from a parent class can add new functionality while still meeting the base specifications required of the parent class. This feature allows for objects inheriting from the same type to be used interchangeably, a mechanism for code re-use and for reconfiguration.

With the development of these mechanisms, OOP was intended to support software re-use and reduce development costs[8]. However, it was subsequently found that, without additional guidance, OOP can result in unworkable software. For example, introduction of C++ into industry exposed major problems. The first truly large-scale project using C++ and OOP was undertaken in 1988 at Mentor Graphics with the decision to completely redesign their CAD application. The project missed its March 1990 deadline by a year. Beta testing sites reported unusually large numbers of errors, and programmers found it difficult to maintain and correct the code.[9]

Object-Oriented Analysis and Design (OOAD) guidelines and principles have subsequently been developed and tested. OOAD does not add additional mechanisms to those provided by OOP; instead, it specifies effective uses of those mechanisms.

One principle of OOAD is the reduction of overall software complexity. This objective can be achieved by reducing extraneous complexity, and by balancing the complexity. For example, an OOP application could theoretically have many simple objects each capable of only one function, or have only one object capable of all their functions; either extreme makes the software appear complex to a developer. OOAD principles require developers to reduce overall complexity in an object-oriented design

Another OOAD principle is the elimination of cyclic dependencies. An example of cyclic dependencies is shown in Figure 2; three objects all have the capability to call each other's methods. In such a situation, the abstraction of having higher- and lower-level objects breaks down, and other objects in the simulation can not be certain whom to contact to get data from, or give data to, the entire aircraft. To a developer new to the simulator, understanding this implementation can be very difficult.

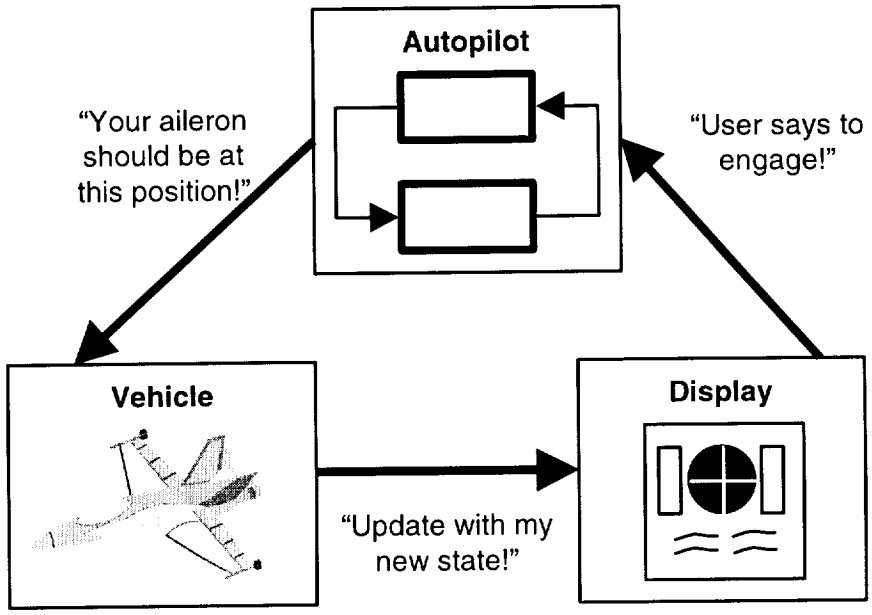

Figure 2. Illustration of cyclic dependencies

Other OOAD principles advise avoiding intrinsically coupled objects. With encapsulation, objects should keep their internal functioning private. However, poorly-designed objects make many of their methods and variables public; in such cases, objects can involve themselves in the internal functioning of other objects. Such internal couplings can result in the violation of abstractions about the functions of objects; as such, modifications to one object might affect a different object in a manner not foreseeable by a programmer. Such situations make the software hard to understand because the control logic for a single process can jump from object to object, and makes testing very difficult because objects cannot be tested independently.

Objects will need to depend on other objects for data. For example, a specialized engine display may require data that can only be provided by a specialized engine model. However, these dependencies can proliferate to such an extent that the use of objects is not reduced to isolated configurations or combinations, as shown in Figure 3. OOAD guidelines help prevent such a proliferation.

In summary, OOP can provide the much needed benefits of understandable, readily modified, easily re-used software. However, these benefits can

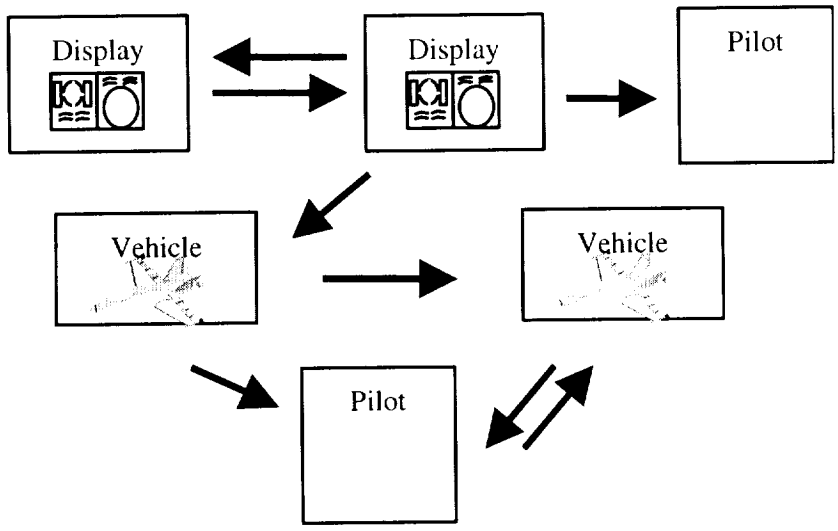

Figure 3. Proliferation of Dependencies Between Objects

only be fully realized if OOAD principles and guidelines are followed. These OOAD stipulations can not be met through low-level programming standards that specify only such items as conventions for naming variables – instead, they require the software architecture to be well-thought-out from its inception.

DESIGN OF THE RECONFIGURABLE FLIGHT SIMULATOR

The Reconfigurable Flight Simulator (RFS) was designed to meet the requirements of a simulator useful for aerospace research and design activities. The architecture was designed in an object-oriented manner with attention to OOAD principles. It is programmed in C++, with graphics capability provided by OpenGL, to facilitate portability across platforms.

Overview of the Simulator Architecture

The RFS is highly modular. The main RFS application does not contain any simulation models. Instead, as shown schematically in Figure 4, the main application provides the run-time support for individual simulation components. This run-time support includes initializing and registering the individual components, and providing communication between them. The main application also contains the interface standards that the components must inherit from.

The components, or plug-in modules, are each stored in a precompiled library that can be loaded by the simulator during run-time. The user can select from a library of available components to configure the simulation as desired for any particular run. Developers can extend the capabilities of the simulator by creating new modules.

The major components of the RFS are shown in Figure 5. Arrows in this diagram represent the access each component has to other objects in the simulation. Cyclic dependencies were avoided by creating a hierarchy within the components; in the dependencies shown, for example, the scenario object can call the four types of objects that are 'underneath' it, the simulation controller objects can call three types of objects, and so on, with the Environmental Controller and Database (ECAD) object at the bottom of the chain. All of the objects are based on a simulation foundation class (SFC). This is represented in the dependency diagram shown in Figure 6.

To provide a dynamic framework, the RFS architecture supports swapping, removing, and loading components in the simulation during run-time. To prevent problems with violation errors (where objects attempt to access components that have been removed) and to ensure that a newly-loaded component is used properly by other objects, a notification system notifies all components when discrete changes occur.

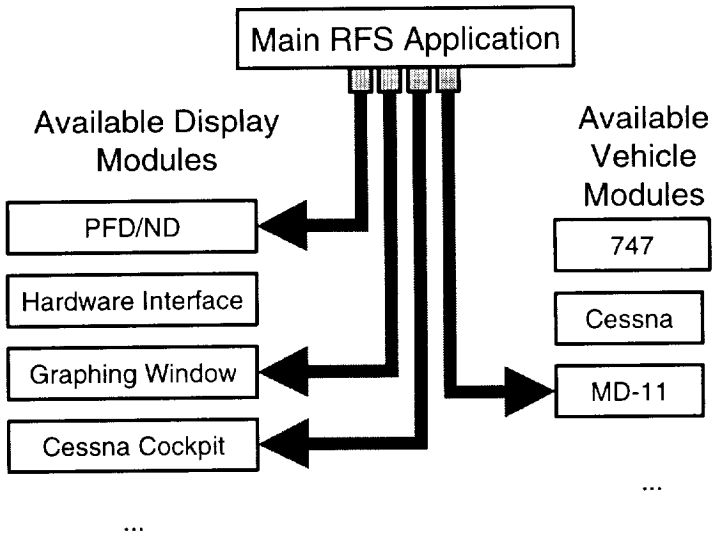

Figure 4. Simulation Access of Components

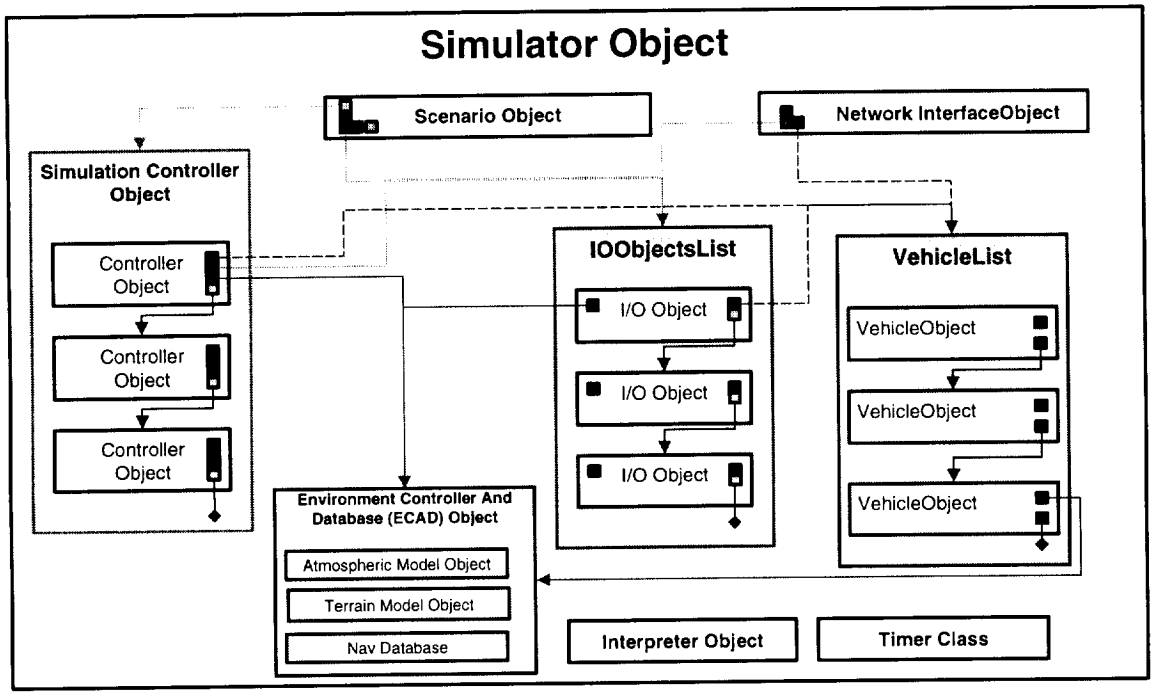

Figure 5. Main RFS Components
(Arrows Represent Access to Other Objects In the Simulation)

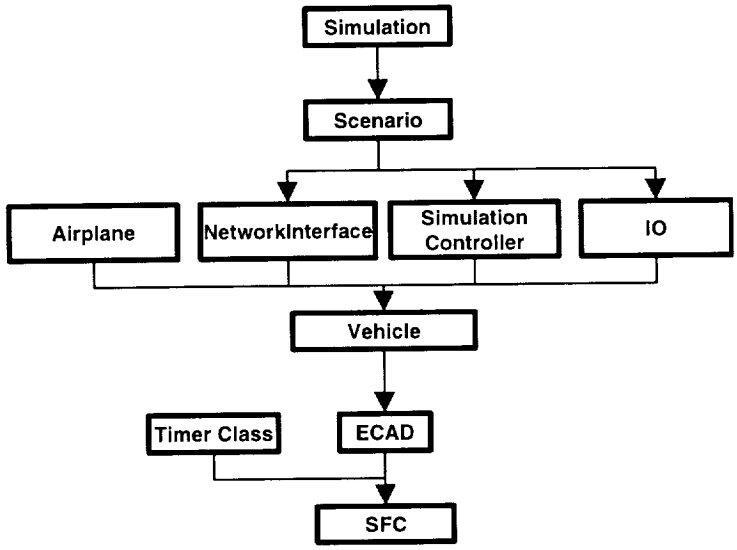

Figure 6. Dependency Diagram of the RFS Standard Interfaces

Components Within RFS

The components of RFS are listed in Table 1, with a brief description of their functions within the simulation. Some of these components provide the functionality within the main RFS application; others establish the standard interface for plug-in modules.

The component-based design of the RFS allows developers to create plug-in modules providing new vehicle models, displays or simulation controllers. Because these components are stored in linked-lists in the

Table 1. Description of Core Simulator Objects

Components Integral to the Main RFS Application	
SimulationObject	This object is the main object in the simulation. The SimuationObject maintains all other objects that will take part in a simulation, which entails establishing communication links between objects in the simulation, and ensuring that all objects are notified when an object is destroyed or is no longer taking part in the simulation. The simulation object is also responsible for implementing the main simulation loop, notifying each component and providing them with the new simulation time each time through the loop.
ScenarioObject	The scenario object maintains the simulation modules. It is responsible for loading and unloading DLL modules and extracting objects from the modules. Once loaded, the scenario object is responsible for placing these objects in their appropriate location.
InterpreterObject	The interpreter object is responsible for providing users with the ability to manipulate objects at the method/attribute level in real-time. This object translates user commands into method calls and attribute queries, accepting text strings that are in a format similar to C++. The interpreter object also provides for remote method invocation when RFS is operating over a network. Method invocations are passed over the network and received by this object, which interprets and dispatches the invocation.
TimerObject	The timer object is responsible for maintaining the simulation time, providing components with a common resource for timing needs. The user can manipulate the simulation time through this object. The timer can operate in many different modes, including real-time by referencing the system clock, or in fast-time.
ECAD	The Environmental Controller and Database object provides the simulation with a database for environmental data, including terrain, wind information, and a navigational aide database. The ECAD further provides axis definitions for common axis systems that will be used by the simulation as well as conversion routines between these systems.
VehicleList	This object maintains all vehicles in the simulation in a collection, which is implemented as a linked-list. Components in the simulation can retrieve particular vehicles by traversing this list. The VehicleList can also distribute certain methods invocations to all vehicles that it contains.
IOList	This object implements a collection of I/O (input/output) objects, implemented as a linked-list. Input/output objects include any component that serves in an I/O capacity. Hardware interfaces, data storage components, pilot input devices, and avionics displays are all considered to be I/O objects in the RFS.
ControllerList	This object implements a collection of simulation controller objects. Simulation controllers are objects that manipulate or control some aspect of the simulation. Examples include ATC controller models, randomly generation of aircraft to place in an air traffic control simulation, and discrete events at scripted times such as mechanical failures.
Base Interface Standards for Components to be Added by Developers	
BaseVehicle	This interface object defines communication for vehicles in the simulation. All vehicles in the simulation, including airplanes and ground vehicles, implement this interface. Components can access common parameters such as position, orientation, and velocity. A baseAircraft interface standard has also been created that inherits from this interface.
BaseIO	This interface defines communication for I/O (input/output) objects. This interface comprises mostly of callbacks invoked when certain simulation events occur (such as starting and stopping the simulation, or the start of each time step).
BaseController	This interface defines communication for simulation controller objects, comprised mostly of virtual callback methods. Simulation controllers must implement this interface.

simulation, the number of each which can be included in a simulator configuration at one time is limited only by the computer hardware on which the simulator is being run. The functionality of the main RFS application can also be extended.

This component-based architecture has several advantages. Developers in different locations and on different projects can create new components and upload them to a central repository; as such, a distributed development environment is possible. Since each module is encapsulated, the developer can work on individual modules without needing knowledge about other components. This facilitates code re-use and reducing the amount of time to tailor the simulation to particular applications. In addition, simulation developers do not require a broad range of expertise. For example, a flight controls designer need only modify an aircraft's dynamic model and flight control design, using established display modules without needing knowledge of computer graphics.

General users also benefit from this architecture, as they can create new simulations from this growing library of modules. Also, a simulation user can store only those components that are needed for their applications, reducing storage requirements.

This paper has focused on how these components are fit into the RFS framework. The specific details of their underlying models are already well covered in the existing literature, ranging from aircraft dynamic models[10] to computer graphics [for example, 11,12]. Also, graphical programming tools, and code generators are increasingly prevalent for these types of components [for example,13,14].

Object Data/Method Extensions

The base objects within the main RFS application define the minimum communication standards for components within the simulation. However, if these standards were the only communication mechanism, then they would constrain the components unnecessarily. For example, a particularly intricate display may require information from an aircraft that is not available through the base vehicle interface.

To extend these base vehicle definitions, the Object Data/ Method Extensions (OD/ME) interface was created. It establishes a generic, simulation-wide mechanism for message and data passing between objects.

All objects in the RFS have OD/ME capabilities built into their base classes, giving them the ability to declare methods and variables to the entire simulation. The variables can be write-able or write-protected. Other components can then access these methods and variables through the OD/ME interface.

OD/ME adds to the simulation the power to use components with arbitrary communication requirements; this corresponds to an ability to include components of arbitrarily high fidelity and detail. However, because it passes through an interpreter, OD/ME access can be an order of magnitude slower than direct access through the base class interfaces.

User Control Over Simulation Runs

The OD/ME interface also provides the user with the ability to access all aspects of the simulation during run-time. The most basic method of access is through Graphical User Interfaces (GUIs), such as data viewer windows and graphing windows; such GUIs can be implemented as components meeting the base IO object interface standards.

In addition, the interpreter object can map strings of characters to specific method invocations or data requests. As such, the user can control the simulator directly through text commands in a command-line window. Commands are entered through the console window and parsed by the interpreter. The parsed commands are stored as character strings in the computer memory, which OD/ME can map into method invocations or data retrieval. In this way, the user can manipulate during run-time any object in the simulation through method invocations and data access.

This interpreted access is a powerful feature of RFS. Interpreter commands were built into RFS that allow the user to control the simulation through the command line window, including the ability to add new components to the simulation, and to list all methods and variables that are available within any active component. The interpreter language is similar in form to C++, so that developers and users will not need to learn a new syntax. Developers can test new components from the command line by executing their methods one-by-one. Interactions between components can be configured during run-time to meet the needs of the task; for example, a button from a joystick can be mapped to any method of any component active in the simulator, so that pressing the joystick button can execute the method.

Command line inputs can also be stored in plain text script files that the interpreter object can access. As such, general users can configure the simulator before run-time through plain text files, without requiring any access to the source code or recompilation. These script files can also contain references to other script files, so that standard configurations can be defined and referred to, reducing the complexity of any individual script file.

Networking

Networking has proven invaluable for performing simulations when the processing load is too great for a single processor, as is common in flight simulations with high-fidelity dynamic models and/or extensive graphics. Networking is an integral part of many contemporary architectures. For example, the LaSRS++ simulator requires that all components access shared memory, which can be passed between machines running the simulation. However, as is common in current simulation architectures, the networking is at a 'lower-level' than the vehicle, making the vehicle object dependent on the networking object. As a result, changes in networking methods or configuration may require modifications to all components that use networking.

The RFS architecture opted to place the networking a higher level then the components, with the result that networking capabilities are transparent to component developers and components do not need to build in special features to access the networking interface (Instead, the burden is placed on the developer of a networking interface to determine the information that is to be passed). Further, the overall architecture is not reliant on a single networking protocol. Currently, a networking component has been developed that uses the High Level Architecture (HLA) standards, but other networking protocols can be created and used.

CONCLUSION

This project developed simulation suitable for use as a general research and design tool. While simulation can bring powerful analysis capabilities to research and design, its use has traditionally been limited by the time and resources required it, and by the use of different simulation architectures in different domains.

The RFS provides a framework that is extremely flexible and extendable to fulfill these needs. Simulation users are provided with near-complete control over the simulation and its components, and can build on existing components to rapidly create new prototypes and solutions.

Through adherence to OOAD principles, the simulation architecture can benefit developers by helping them avoid the common pitfalls of object oriented development while delivering a system that is easy to understand, use, and adapt for a particular purpose. Further, the ability to run on desktop machines without specialized and expensive hardware provides instant accessibility to users and developers on a well-known and easy to use platform.

The RFS has proven to be a stable and scalable platform for development. Designers benefit from a library of components that facilitate development, such as real-time monitoring and graphing components, as well as features such as the programmable real-time environment to provide extreme control over all objects in the simulation during run-time. The RFS has been used for a variety of simulations, from fast-time large-scale simulations of air traffic control[15], to single-aircraft pilot-in-the-loop studies of cockpit systems[16].

Through its flexible architecture, the RFS provides a single software simulation solution for the entire design process. As a design matures, it requires a simulation that can adapt to new requirements and higher fidelity. The RFS provides a flexible and scalable framework for early design work which can adapt as more detailed and accurate simulations are required. For conceptual design and exploratory prototyping, component reuse and modification of existing capabilities can provide a fast solution. As the design progresses and requirements grow, the RFS adapts by allowing the development and integration of higher fidelity components, hardware interfaces, and other tools to expand the simulation as requirements grow, such as HLA networking to distribute processing load to multiple processors.

The availability of simulation architectures suitable for a wide-range of research and design activities (such as RFS) can be reasonably predicted to have a dramatic impact on aerospace research. Research practices can benefit from a simulation that allows for different researchers to share components and test each other's conclusions. Likewise, the general availability of a simulation may enable small research groups to apply better simulation tools then previously possible.

Such simulation architectures may also have a dramatic impact on aerospace design practices. Designers' needs in a simulation mature as the design proceeds[3,17]. A simulation that can progress in maturity and be used in every stage of the design can serve not only as an analysis tool, but as the repository of design knowledge, to which designers add increasingly detailed specifications in the form of simulation components.

REFERENCES

[1] Tuohy, S.T. "An Integrated Vehicle Simulation For the Development and Validation of a Commercial Reusable Launch Vehicle" *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Boston MA.

[2] Totah, J.J. and Kinney, D.J. (1998) "Simulating Conceptual and Developmental Aircraft" *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Boston MA.

[3] Norlin, K.A. (1995) "Flight Simulation Software at NASA Dryden Flight Research Center" *AIAA Modeling and Simulation Technologies Conference and Exhibit.*

[4] Walker, W.H. and Kircher, R.C. (1997) "Modern Component Methods for Air Transport Engineering Simulation Software" *AIAA Modeling and Simulation Technologies Conference and Exhibit*, New Orleans LA.

[5] Leslie, R.A., et al (1998) "LaRS++ An Object-Oriented Framework for Real-Time Simulation of Aircraft" *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Boston MA.

[6] Khoshafian, S. and Abnous, R. (1995) *Object Orientation: Concepts, Analysis & Design, Languages, Databases, Graphical User Interfaces, Standards*, Wiley, New York NY.

[7] Paterson, D.J. (1998) "Simulation Middleware Object Classes (SMOC) for Simulation and Modeling Applications" *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Boston MA.

[8] Alagic, S. et al (1996) "Object-Oriented Flight Simulator Technology" *AIAA Modeling and Simulation Technologies Conference and Exhibit*, San Diego CA.

[9] Lakos, J. (1996) *Large Scale C++ Software Development*, Addison-Wesley, Reading MA

[10] Stevens, B. and Lewis,F. (1992) *Aircraft Control and Simulation*, Wiley, New York NY

[11] Foley, J. et al. (1995) *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading MA

[12] Woo, M. et al. (1997) *OpenGL 1.2 Programmers Guide*, Addison-Wesley, Reading, MA.

[13] Ippolito, C.A. and Pritchett, A.R. "SABO: A Self-Assembling Architecture for Complex System Simulation", *The 38th AIAA Aerospace Sciences Meeting and Exhibit*, Reno NV.

[14] Robins, A. et al (1998) "Commercial Visual Programming Environments: One Step Closer to Real Simulation Reuse" *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Boston MA.

[15] Pritchett, A.R., Lee, S.M., Huang, D. and Goldsman, D. (2000) "Hybrid-System Simulation for National Airspace System Safety Analysis", *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Denver CO.

[16] Pritchett, A.R. and Yankosky, L.J. (2000) "Pilot Performance at New ATM Operations: Maintaining In-Trail Separation and Arrival Sequencing", *AIAA Guidance, Navigation and Control Conference*, Denver CO.

[17] Williams, G.B. (1997) "The Boeing Commercial Airplane Engineering Simulation, New Airplane Project, Management Observations" *AIAA Modeling and Simulation Technologies Conference and Exhibit*, New Orleans LA.